IN TOUCH: A GRAPHICAL USER INTERFACE DEVELOPMENT TOOL

P. Rosner, M. Magennis and A. Newman
Queen Mary and Westfield College, London
and London HCI Centre

BACKGROUND

Recent tools and architectures for the production of user-interface software have addressed both the requirement for quality user-interfaces and also the needs of the designer in producing such interfaces.

User Interface Management Systems (UIMSes) are based on an architecture that separates user-interface from application by providing an intermediate dialogue control layer. The aim is to enable an iterative design/prototyping cycle, allowing different user-interfaces to be tried out on the same application foundations. A UIMS also provides a representation for overall dialogue structure abstracted from presentation details - thus allowing for a closer correlation with the task being performed by the user. This dialogue representation is not just a diagram or notation to specify a later coding stage, but is compiled or interpreted along with the user-interface and application components. Executable code, with suitable run-time support from the UIMS, is thus generated. In this sense a UIMS provides an *executable specification* of an interactive application.

UIMSes, however, have not been used as extensively as expected. The main problem has been that the dialogue representations (e.g. state transition diagrams, grammars, event-based languages) become too unwieldy for anything beyond simple applications. In particular, direct manipulation user interfaces such as the Macintosh desktop do not lend themselves easily to the split between application and user-interface embodied in the traditional UIMS architecture. Interactive systems have been constructed, in practice, using more dispersed architectures, user interface toolkits and object-oriented techniques. Whilst these successfully overcome the limitations of the UIMS architecture, the overall structure of the application and the ability to relate it to the user's task are lost in the tangle of communicating software entities.

OVERVIEW OF IN TOUCH

In Touch, being developed by the London HCI Centre (Alvey Project MMI/151), has been designed for the class of non-direct-manipulation graphical user interfaces, typical examples being in public access, touch-screen systems. For such systems a graphical representation of overall dialogue structure, close to the traditional UIMS approach, is still appropriate. The representation used is based on state-transition diagrams with the important enhancements of nested state hierarchies, event sharing and parallel states. These help to overcome the visual complexity encountered in state transition diagrams and help the designer to denote structured and parallel user-dialogues in a visual form. A standard composite state representing a user-resource (e.g. a screen-based keypad) can be positioned alongside an appropriate state in the basic dialogue structure to denote the desired availability of the resource to the user within the dialogue.

In Touch also supports the design process by providing *animation* and *simulation* facilities. *Animation* means that the graphical representation of the dialogue can be presented to the designer (and even possibly the commissioner or user of the system) alongside the running interface; its various components highlight or dehighlight to indicate the state of the dialogue. *Simulation* means that the designer can interact directly with the diagram to simulate user actions or system events, even before the associated sections of the interface have been constructed (as in many design situations).

Another aim of In Touch has been to enable both top-down and bottom-up design, again a feature of many design situations, as described by Hartson and Hix (1). The top-down approach means inserting user-interface elements into a diagram of the basic dialogue structure. The bottom-up approach means the automatic construction of elements in the dialogue diagram as interface components are introduced by interactive means into the interface area. Only the top-down design style is supported in the current In Touch implementation, although the architecture allows for the incorporation of both styles.

In Touch has been designed to produce code executable on a range of target hardware (distinct from that of the development environment), implying portability of generated code and a minimal run-time kernel, which is the only component that needs to be target-specific.

ARCHITECTURE OF IN TOUCH

Fig 1 shows the architecture of In Touch. The run-time environment is an interpreter for tokenised ScreenScript, a language developed by the London HCI Centre for the generation of presentation-quality graphical output and the provision of sensitive screen areas for user input. Procedures, collectively termed the *State Engine*, have been written in ScreenScript enriching its state/event mechanism with a hierarchical structure as described in the next section. The designer can generate an interface either by using the graphical construction tool or by programming directly in ScreenScript.

The representation produced by the graphical tool combines graphical elements with fragments, termed *scripts*, of ScreenScript code. A translator converts this into a complete ScreenScript source program with suitable invocations of State Engine procedures. The compiler converts ScreenScript source code into tokenised code. This code, when executing, reacts to events from either the user or the system, generates graphical output, and invokes application routines. The types of events that cause transitions are either primitive, intrinsic to the basic ScreenScript implementation, or else synthetic, which means they are generated by scripts. There can be several instances of a primitive event type concerned with pointer activity (e.g. pointer up); these instances are specified by the sensitive screen area in which the pointer activity occurs.

The executing code can respond to pseudo-events produced by interaction with the graphical representation, enabling simulation of the dialogue. It can also issue events that indicate states being opened and closed, enabling animation of the graphical representation.

In the current implementation both the development and run-time environments are based on the Apple Macintosh™, the former running under SuperCard™.

STATECHARTS

Figs 2 to 5 show part of the representation for a bank 'hole in the wall' cashpoint simulation. The diagrams were generated using the graphical tool for producing In Touch diagrams. The cashpoint has a display area, arrow buttons either side of it for option choice, and a keypad for entry of personal identity number and cash amount. These diagrams will be used to illustrate the dialogue representation. (For the purposes of black and white illustration, the coloured arrows, blobs and lines have been replaced by patterned versions.)

The graphical state-transition representation used by In Touch, and embodied in the State Engine, is derived from Harel's (2) statechart formalism. *States* are represented by boxes on the diagram. Arcs emanating from state boxes may be labelled with an *event* or unlabelled. An arc has an arrow-head that is coloured or uncoloured. If a labelled arc points to another state, then on occurrence of the event, a *transition* to this new state occurs. A *script* can be associated with, and will be executed as part of, the transition. A labelled arc with an uncoloured arrow head not pointing to another state signifies an *event-action*, with an associated script executed on occurrence of the event but the current state remaining unchanged. Entry and exit scripts can also be associated with a state. Scripts are written in ScreenScript and can, amongst other things, display graphics to the screen, transmit further events, or call application procedures. In Fig 4, for example, a synthetic timeout (*t/o*) event causes a transition between each state in the sequence. The entry procedure for each state displays graphics on the screen and, after generating a suitable delay, posts event *t/o*.

A state may be primitive or composite. A diagram represents either the top level of the dialogue or else the expansion of a composite state. An expansion diagram is invoked when an arc pointing to its composite state representation is traversed. Thus the *welcome* diagram of Fig 4 is an expansion of the *welcome seq* composite state of Fig 3. A *blob* can be attached to any state within a diagram. The colour of the arrow-head in the invoking diagram is matched with the colour of a blob in the invoked diagram to determine the initial state to be entered on invocation. Different colours can thus be used to allow different initial states within a sub-diagram. For example the composite state *service choice* shown in Fig 3 has two entry points, one denoted by the black arrow-head and one by the coloured arrow-head. These correspond to the black and coloured blobs of Fig 5. A diagram always has a default black blob; the default colour of an arrow-head is also black.

An arc with a coloured or black arrow-head pointing neither to another state nor to the edge of the diagram, indicates a connection to a state in the same diagram; the colour of the arc's arrow-head and the colour of the blob attached to the destination state have to match - a useful mechanism for avoiding unwanted lines cluttering the diagram. In Fig 3 the arrow-head at the end of each row indicates a connection, via the blob of the same colour, to the state at the start of another row. Another means of reducing clutter is a *connector box* which is simply a means of joining an arc to its destination. In Fig 3 a connector box connects the transition labelled *wrong* from state *check pin* to state *enter pin*, and in Fig 5 a connector is used to join three arcs such that all exit the diagram.

A labelled arc from a composite state box siginifes a *generic event* for that state; all its open child states (which may extend several levels down the hierarchy) are closed before the transition is performed. In Fig 3 the generic event *card in* causes exit from the composite state *welcome seq*, irrespective of which child state (see Fig 4) may be open at the time. An unlabelled arc from a composite state signifies a transition from a particular child state to a state outside the composite state. In the expansion of the composite state, the exiting arc extends to the border of the diagram. Again a colour matching system is used to determine the destination state: the colour of the exiting arc's arrow-head in the subdiagram is the same as the stem colour of the arc from the corresponding composite state in the parent diagram. In Fig 3 there is an unlabelled arc with a coloured stem from the composite state *service choice*. This corresponds, in Fig 5, to the arc, whose arrow-head has same colour, pointing to the edge of the diagram (i.e. one of the transitions *statement*, *chk bk* or *balance*).

Special states called H^* and H can be used as the initial states within a diagram. These enable control to be passed to either the lowest level primitive state or else the composite state active when the diagram was last exited (useful for denoting, for example, the return of control after an error or help sequence). We have also augmented Harel's formalism through the provision of *modal* states. Whilst a modal state is active, the generic transitions in ancestor states are inhibited. Modal states have been found necessary for representing, for example, modal dialogue boxes.

The representation allows parallel states within a diagram. If states (which may be composite) are *anded* together, then a transition from any one of them will cause all its and-states (and any associated sub-hierarchies) to be closed as well. In our graphical representation, parallel states are denoted by overlapping state-boxes. In Fig 2, the top level diagram, the *main control* state has two associated and-states. One is an event generator *keypad*, which interprets key press events in different sensitive areas and broadcasts an event with the key value as parameter. Similarly the *arrows* event generator interprets arrow key press events and transmits the appropriate event such as 'middle arrow, right'. Since these two event generators are and-states of *main control*, they will be active throughout its life. In Fig 3, the states *confirm arrows* and *sc arrows* translate events sent in the top level *arrows* state, converting them to synthetic events recognized by their own and-states states. Similarly the *number edit* state converts a sequence of top level keypad events into a synthetic *done* event, sent to its and-state. The *welcome sequence* has an and-state *card-in detector*. This is an event generator that generates the synthetic *card in* event when the *insert card* button is pressed.

The simulation described here is of a bank cash machine with hard (as opposed to screen) buttons for the keypad and arrows. However touch screen buttons that appear only when relevant to the dialogue could easily be represented by simply removing them as and-states of the top level and incorporating them as and-states of the appropriate lower level state.

The diagrams as constructed by the graphical editor can coexist on the screen alongside the complete or partially constructed user-interface itself. As the user-interaction progresses, states within a diagram highlight and dehighlight, and subdiagrams representing composite states appear and disappear. In addition the designer can press the event names on transitions or event actions in order to simulate events for portions of the interface that have not yet been written.

CURRENT STATE OF DEVELOPMENT AND FUTURE WORK

The In Touch project is still under development. The ScreenScript development environment is available as a stand-alone system and the first versions of the State Engine and Graphical editor are also complete. A prototype version of In Touch demonstrates animation capabilities. The full facilities for animation and simulation and the translator from the graphical representation to the full ScreenScript source program are yet to be implemented. Future plans include a re-implementation of the state engine as ScreenScript primitives to improve performance; versions of the run time ScreenScript kernel on the IBM PC/compatibles and X11; versions of the ScreenScript and In Touch development environments under Windows 3 on IBM PC/compatibles; and in the longer term a version of these development environments under X11. Also among the longer term plans are facilities to support the 'bottom up' design approach.

REFERENCES

(1) Hartson, R. and Hix, D. 'Towards Empirically Developed Methodologies', in Human Computer Interface Development, Academic Press 1989.

(2) Harel, D. 'On Visual Formalisms', Communications of the ACM, May 1989, Vol 31 Number 5.

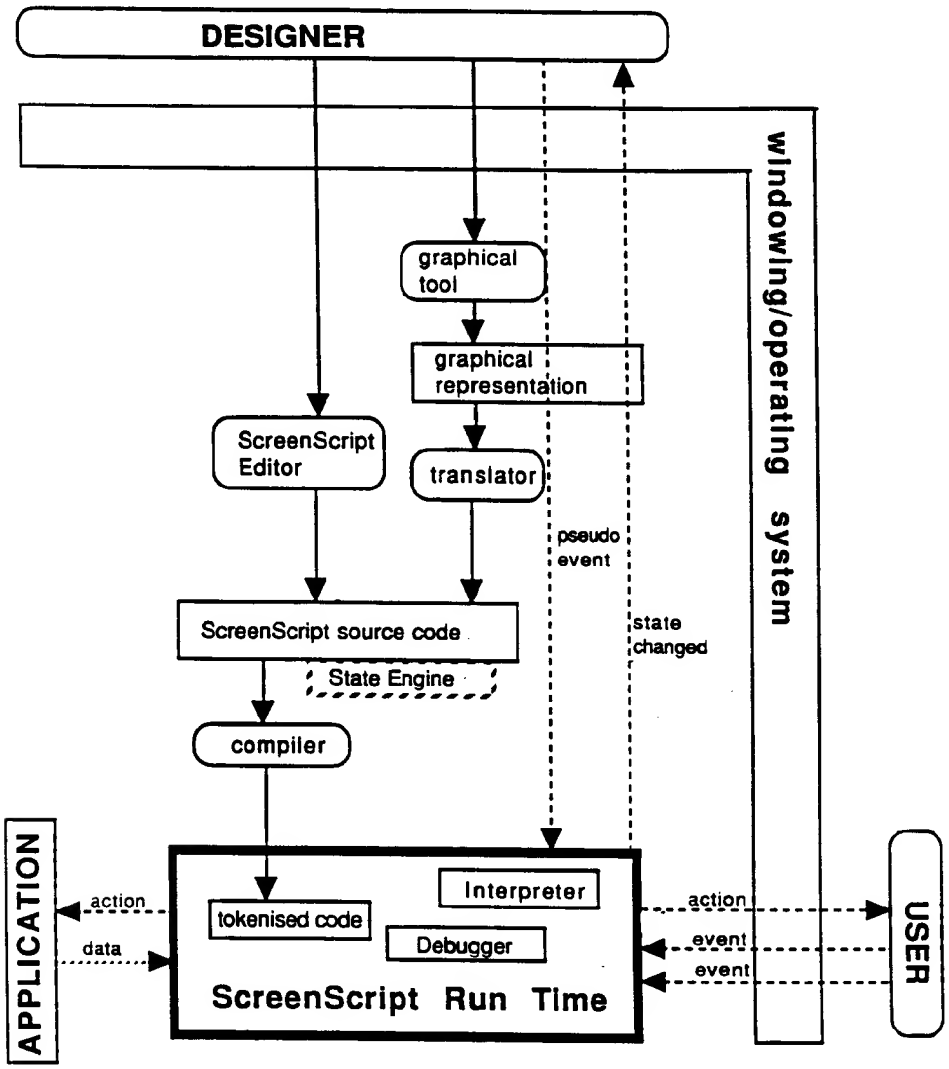

Fig 1 Architecture of In Touch

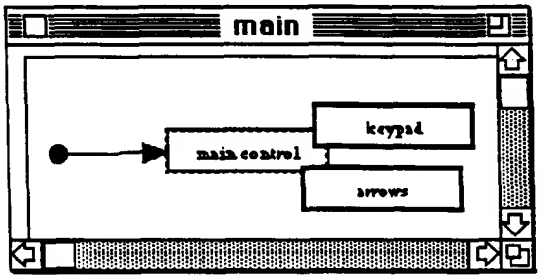

Fig 2 Cashpoint top level

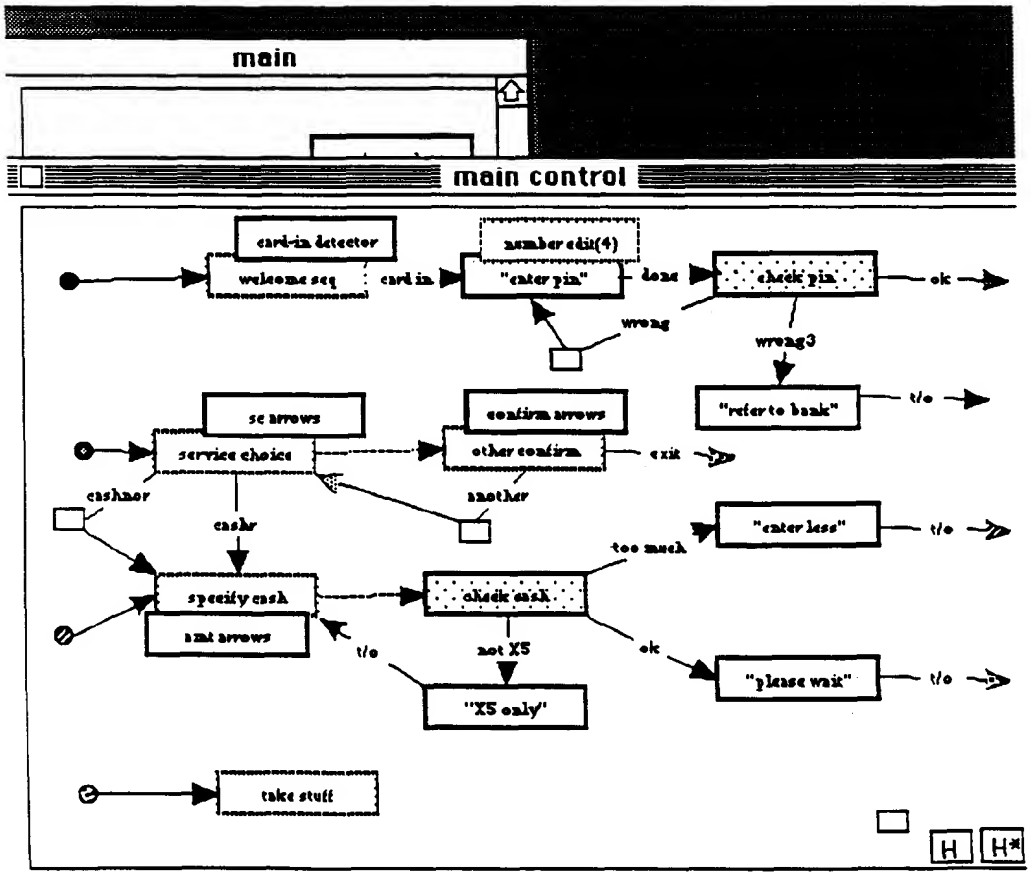

Fig 3 Main Control

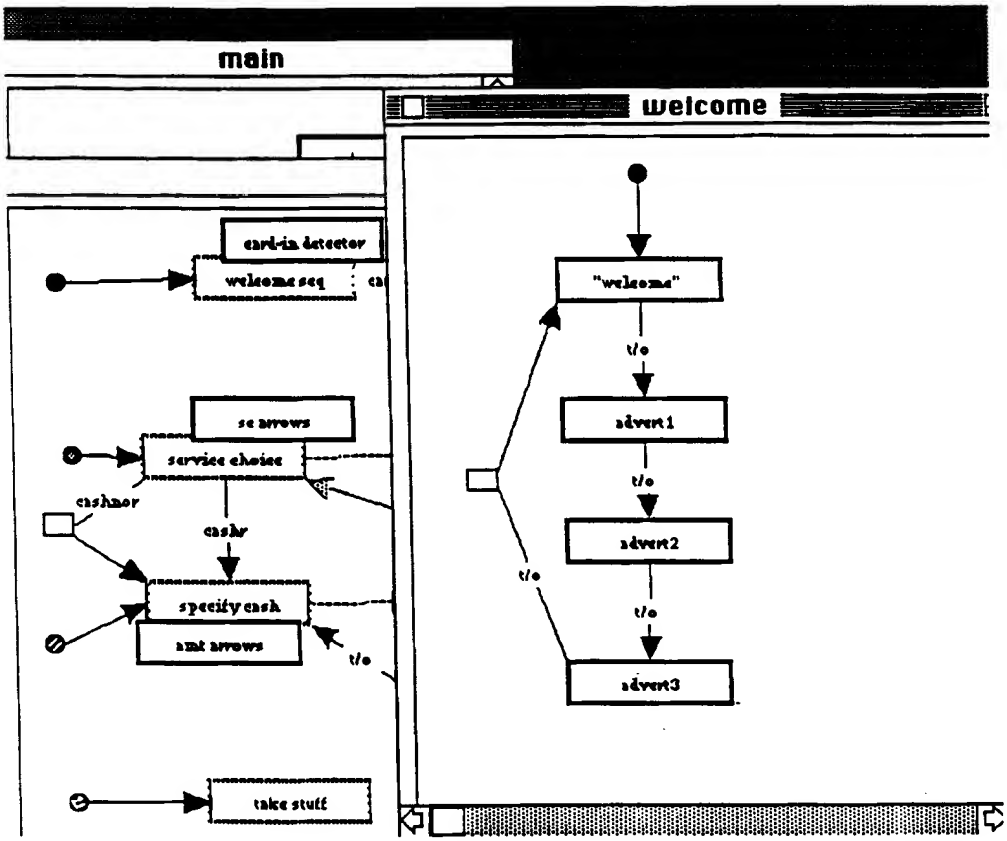

Fig 4 The Welcome Sequence

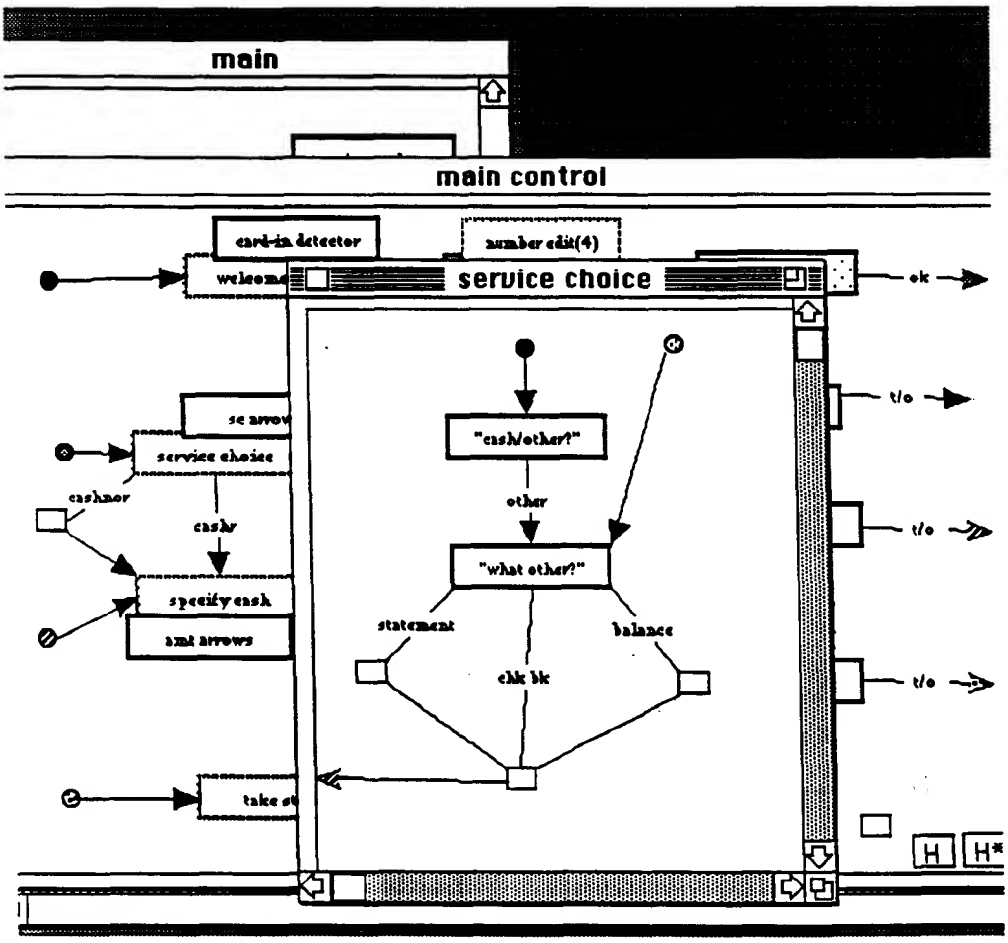

Fig 5 Service Choice